



# Spring Security

# About me



## Trayan Iliev

- CEO of IPT – Intellectual Products & Technologies  
<http://www.iproduct.org>
- Oracle® certified programmer 15+ Y
- end-to-end reactive fullstack apps with [Java](#), [ES6+](#), [TypeScript](#), [Angular](#), [React](#) and [Vue.js](#)
- 12+ years IT trainer: [Spring](#), [Java EE](#), [Node.js](#), [Express](#), [GraphQL](#), [SOA](#), [REST](#), [DDD](#) & [Reactive Microservices](#)
- Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker
- Organizer RoboLearn hackathons and IoT enthusiast

# Where to Find The Code and Materials?

Java Academy projects and examples are available @GitHub:

<https://github.com/iproduct/java-spring-academy-2022.git>

# Security Basic Concepts

- Authentication
- Authorization
- Data integrity
- Confidentiality
- Non-repudiation
- Auditing
- Quality of Service
- Role
- Realm
- User
- Group
- Principal

# Key Java Security APIs - I

- **Java Authentication and Authorization Service (JAAS)** – defines an extensible model for adding new pluggable authentication and authorization modules (PAM)
- **JavaGeneric Security Services (Java GSS-API)** – token-based API for secure exchange of messages unifying access to different security mechanisms such as Kerberos
- **Java Cryptography Extension (JCE)** – provides ability to encrypt, generate, and agree upon keys of symmetric, asymmetric, block and stream cyphers.

## Key Java Security APIs - II

- [Java Secure Sockets Extension \(JSSE\)](#) – provides SSL and TLS transport layer security protocols implementation in Java, allows messages encryption, server (and optionally client) authentication, securing messages integrity.
- [Simple Authentication and Security Layer \(SASL\)](#) – standard for defining an authentication protocol, defining a security layer between client and server applications, providing a framework for implementing concrete authentication mechanisms.

# Security Approaches

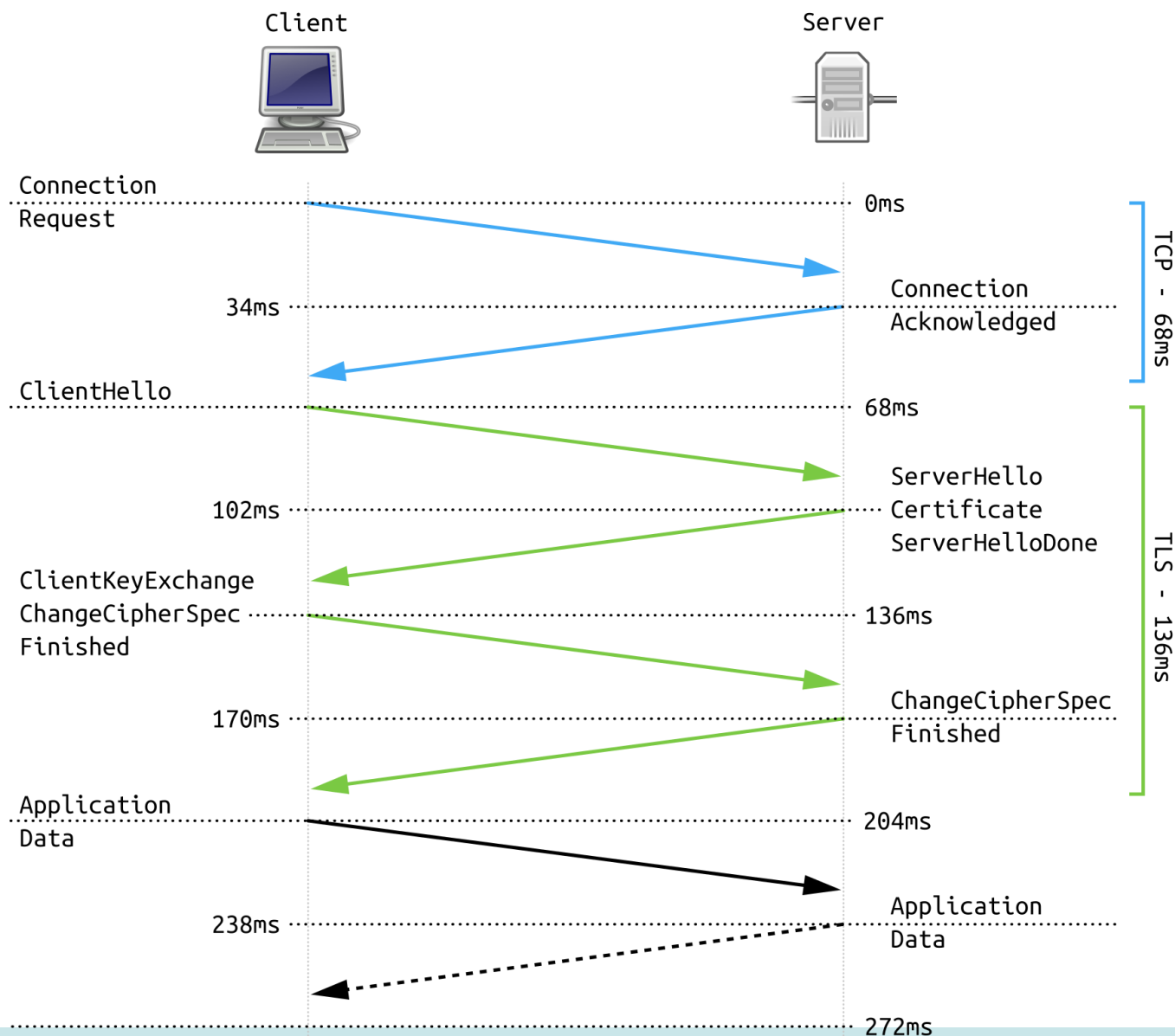
- **Application level security** – security settings specific for concrete application, can not be easily transferred to other applications and environments – secures data while inside application
- **Transport layer security** - HTTPS using SSL / TLS
- **Message level security** – using SOAP with Attachments for message exchange, encrypting message part when confidential

# Transport Layer Security (TLS)

- **Transport Layer Security (TLS)** is a cryptographic protocol designed to provide communications security over a computer network. The protocol is widely used in applications such as email, instant messaging, and voice over IP, but its use in securing HTTPS remains the most publicly visible.
- The **TLS protocol** aims primarily to provide **cryptography**, including **privacy (confidentiality)**, **integrity**, and **authenticity** through the use of **certificates**, between two or more communicating computer applications. It runs in the **application layer** and is itself composed of two layers: the **TLS record** and the **TLS handshake protocols**.
- **TLS** is a proposed **Internet Engineering Task Force (IETF) standard**, first defined in 1999, and the current version is **TLS 1.3**, defined in August 2018.
- **TLS** is the successor of the now-deprecated **Secure Sockets Layer (SSL)**.



# Simplified TLS 1.2 Handshake



# Using Self-Signed Certificates

- `keytool -genkeypair -alias springboot -keyalg RSA -keysize 4096 -storetype JKS -keystore springboot.jks -validity 3650 -storepass changeit`
- `keytool -genkeypair -alias springboot -keyalg RSA -keysize 4096 -storetype PKCS12 -keystore springboot.p12 -validity 3650 -storepass changeit`
- `keytool -list -v -keystore springboot.jks`
- `keytool -list -v -keystore springboot.p12`
- `keytool -importkeystore -srckeystore springboot.jks -destkeystore springboot.p12 -deststoretype pkcs12`
- `keytool -import -alias springboot -file myCertificate.crt -keystore springboot.p12 -storepass password`
- `keytool -export -keystore springboot.p12 -alias springboot -file myCertificate.crt`

# Spring Boot SSL Configuration

server.ssl.key-store: classpath:springboot.p12

server.ssl.key-store-password: changeit

server.ssl.key-store-type: pkcs12

server.ssl.key-alias: springboot

server.ssl.key-password: changeit

server.port=8443

# Spring Security

- Spring Security is a powerful and highly customizable **authentication** and **access-control framework**. It is the de-facto standard for securing Spring-based applications.
- Spring Security is a framework that focuses on providing both **authentication** and **authorization** to Java applications. Like all Spring projects, the real power of Spring Security is found in how easily **it can be extended to meet custom requirements**.

# Spring Security Features

- Comprehensive and extensible support for both [Authentication](#) and [Authorization](#)
- Protection against attacks like [session fixation](#), [clickjacking](#), [cross site request forgery](#), etc.
- [Servlet API](#) integration
- Optional integration with [Spring Web MVC](#)
- Integrations - [Spring Data](#), [Jackson](#), [Cryptography](#), etc.

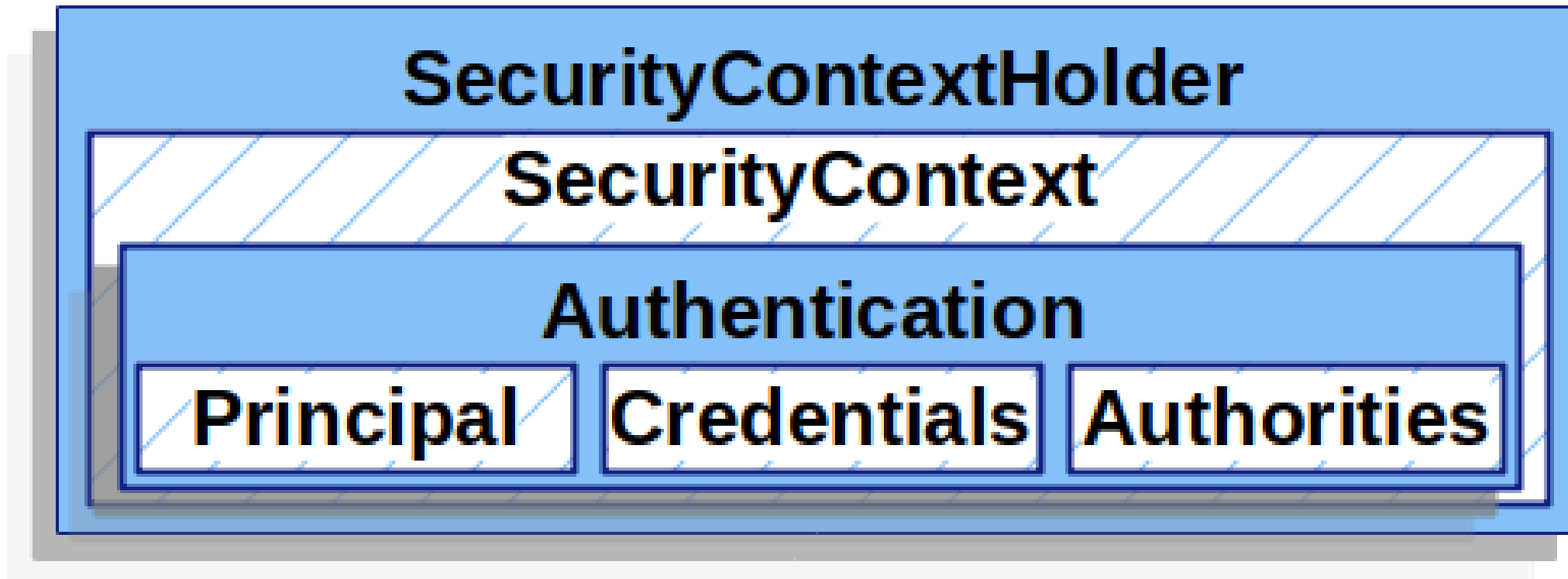
# Authentication

- Spring Security provides comprehensive support for authentication. Authentication is how we verify the identity of who is trying to access a particular resource. A common way to authenticate users is by requiring the user to enter a username and password. Once authentication is performed we know the identity and can perform authorization.
- Spring Security provides built in support for authenticating users. This section is dedicated to generic authentication support that applies in both Servlet and WebFlux environments.

# Authentication Mechanisms Supported by Spring Security

- [Username and Password](#) - how to authenticate with a username/password
- [OAuth 2.0 Login](#) - OAuth 2.0 Login with OpenID Connect and non-standard OAuth 2.0 Login (i.e. GitHub)
- [SAML 2.0 Login](#) - SAML 2.0 Log In
- [Central Authentication Server \(CAS\)](#) - Central Authentication Server (CAS) Support
- [Remember Me](#) - how to remember a user past session expiration
- [JAAS Authentication](#) - authenticate with JAAS
- [OpenID](#) - OpenID Authentication (not to be confused with OpenID Connect)

# Spring Security Architecture





# Spring Security Architecture – Main Parts

- **SecurityContextHolder** - The SecurityContextHolder is where Spring Security stores the details of who is authenticated.
- **SecurityContext** - is obtained from the SecurityContextHolder and contains the Authentication of the currently authenticated user.
- **Authentication** - Can be the input to AuthenticationManager to provide the credentials a user has provided to authenticate or the current user from the SecurityContext.
- **GrantedAuthority** - An authority that is granted to the principal on the Authentication (i.e. roles, scopes, etc.)
- **AuthenticationManager** - the API that defines how Spring Security's Filters perform authentication.

# Spring Security Architecture – Main Parts II

- **ProviderManager** - the most common implementation of AuthenticationManager.
- **AuthenticationProvider** - used by ProviderManager to perform a specific type of authentication.
- **Request Credentials with AuthenticationEntryPoint** - used for requesting credentials from a client (i.e. redirecting to a log in page, sending a WWW-Authenticate response, etc.)
- **AbstractAuthenticationProcessingFilter** - a base Filter used for authentication. This also gives a good idea of the high level flow of authentication and how pieces work together.

# Authentication

- The **Authentication** serves two main purposes within Spring Security:
  - An input to AuthenticationManager to provide the credentials a user has provided to authenticate. When used in this scenario, isAuthenticated() returns false.
  - Represents the currently authenticated user. The current Authentication can be obtained from the SecurityContext.
- The **Authentication** contains:
  - **principal** - identifies the user. When authenticating with a username/password this is often an instance of UserDetails.
  - **credentials** - often a password. In many cases this will be cleared after the user is authenticated to ensure it is not leaked.
  - **authorities** - the GrantedAuthoritys are high level permissions the user is granted. A few examples are roles or scopes.

# Setting SecurityContextHolder

```
SecurityContext context = SecurityContextHolder.createEmptyContext();  
Authentication authentication =  
    new TestingAuthenticationToken("username", "password", "ROLE_USER");  
context.setAuthentication(authentication);  
  
SecurityContextHolder.setContext(context);
```

---

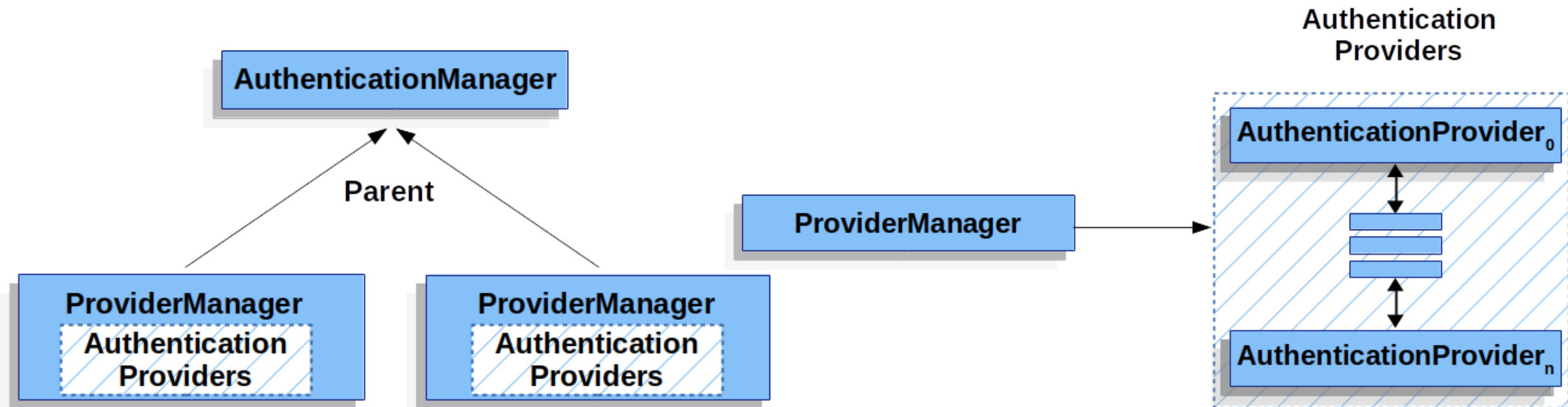
```
UserDetails user;  
SecurityContext context = SecurityContextHolder.createEmptyContext();  
Authentication authentication =  
    new UsernamePasswordAuthenticationToken(user, "password", "ROLE_USER");  
context.setAuthentication(authentication);  
  
SecurityContextHolder.setContext(context);
```

# Accessing SecurityContext Authentication Programmatically

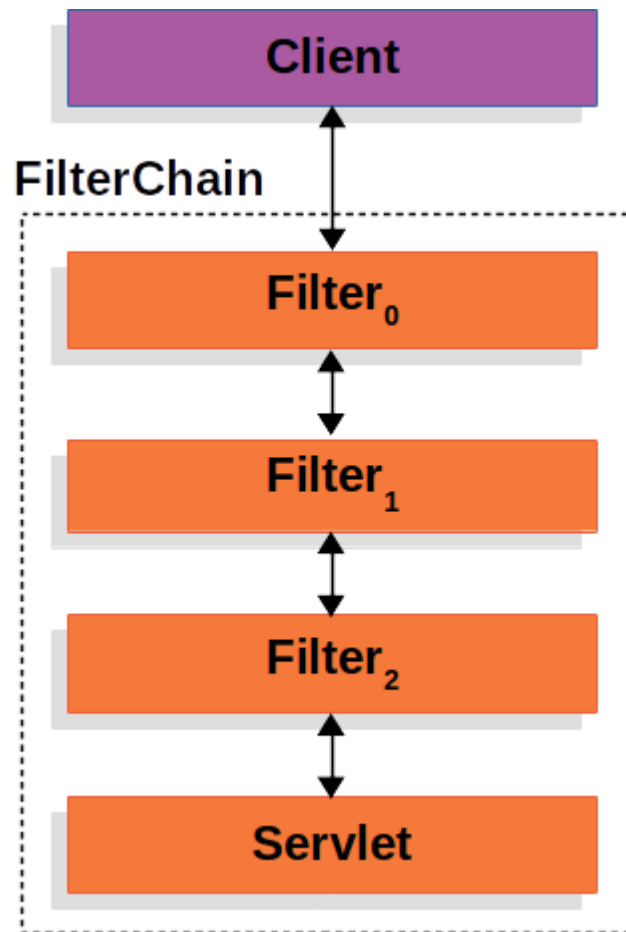
```
SecurityContext context = SecurityContextHolder.getContext();  
Authentication authentication = context.getAuthentication();  
String username = authentication.getName();  
Object principal = authentication.getPrincipal();  
Collection<? extends GrantedAuthority> authorities = authentication.getAuthorities();
```

# ProviderManager

- **AuthenticationManager** is the API that defines how **Spring Security's Filters** perform authentication.
- **ProviderManager** is the most commonly used implementation of **AuthenticationManager**. **ProviderManager** delegates to a List of **AuthenticationProviders**. Each **AuthenticationProvider** has an opportunity to indicate that authentication should be **successful**, **fail**, or indicate it **cannot make a decision** and allow a downstream **AuthenticationProvider** to decide.

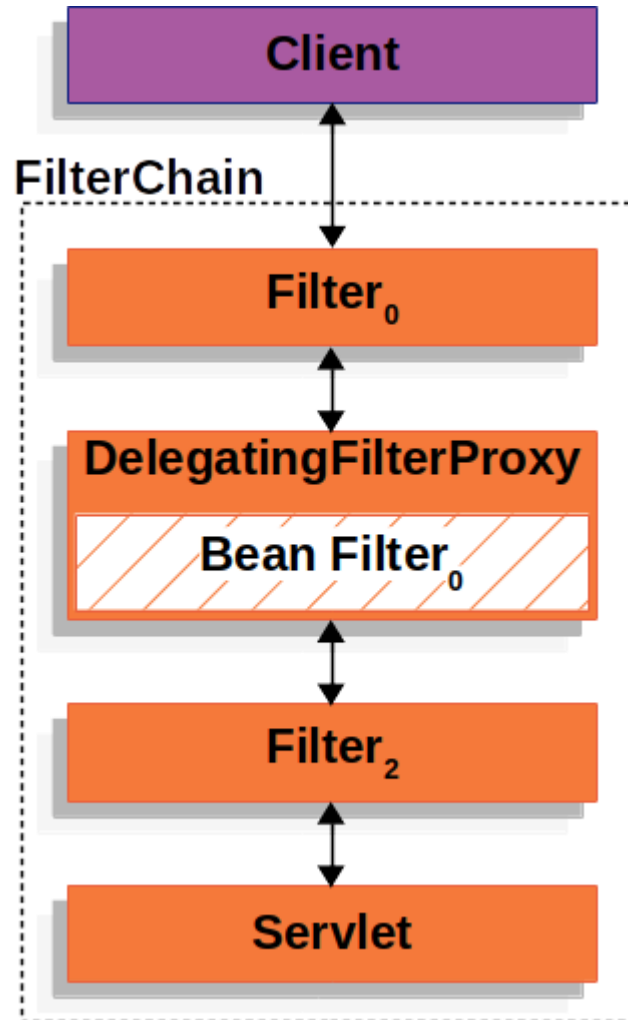


# Spring Security Architecture in Web Servlet Applications



```
public void doFilter(ServletRequest request,  
ServletResponse response, FilterChain chain) {  
    // do something before the rest of the application  
    chain.doFilter(request, response);  
  
    // invoke the rest of the application  
    // do something after the rest of the application  
}
```

# DelegatingFilterProxy

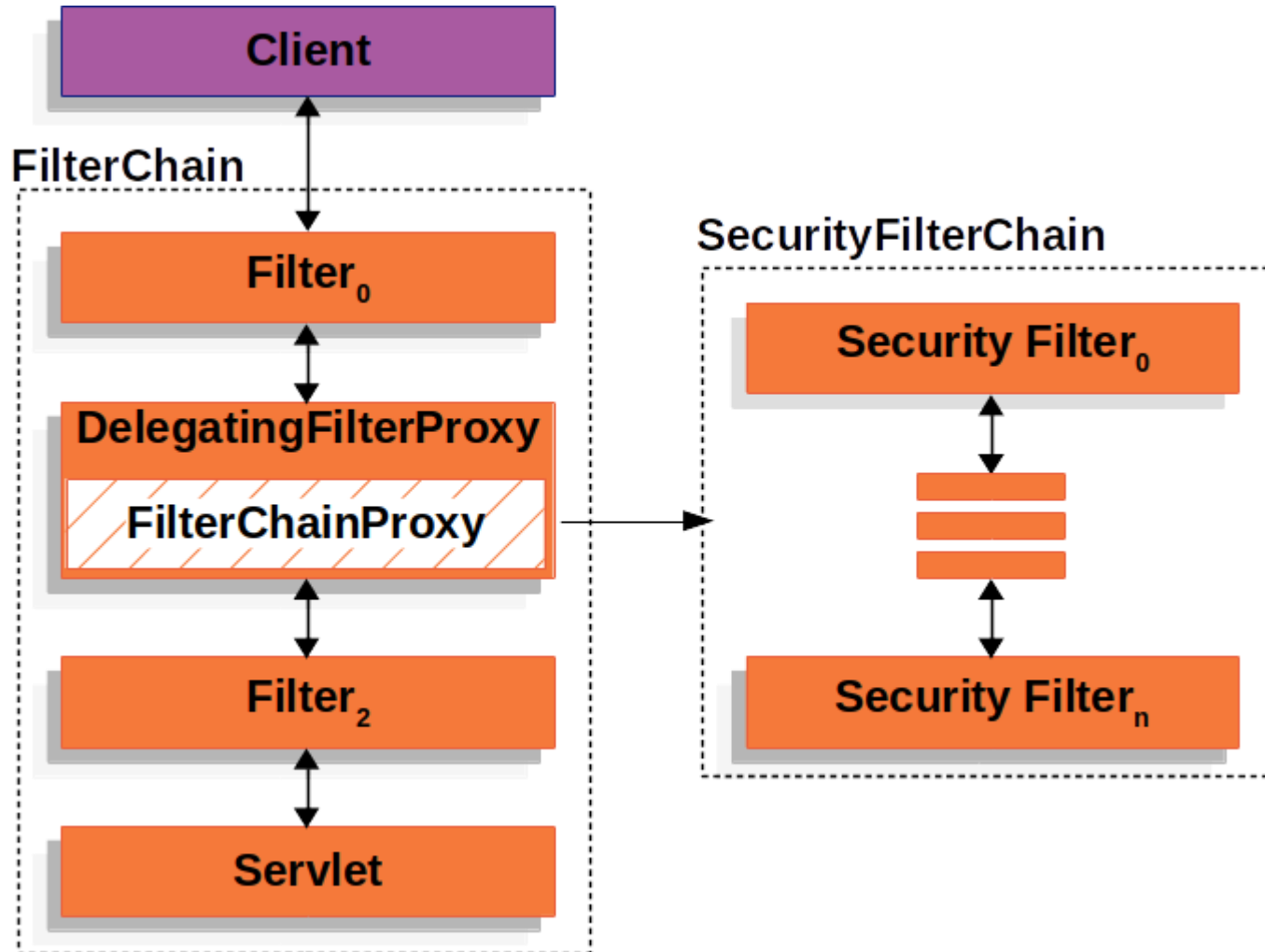


```
public void doFilter(
```

```
    ServletRequest request, ServletResponse response, FilterChain chain) {  
        // Lazily get Filter that was registered as a Spring Bean  
        // For the example in DelegatingFilterProxy delegate is an instance  
        // of Bean Filter0  
        Filter delegate = getFilterBean(someBeanName);  
        // delegate work to the Spring Bean  
        delegate.doFilter(request, response);  
    }
```



# FilterChainProxy



# Security Filters - I

- ForceEagerSessionCreationFilter
- ChannelProcessingFilter
- WebAsyncManagerIntegrationFilter
- SecurityContextPersistenceFilter
- HeaderWriterFilter
- CorsFilter
- CsrfFilter
- LogoutFilter
- OAuth2AuthorizationRequestRedirectFilter
- Saml2WebSsoAuthenticationRequestFilter
- X509AuthenticationFilter
- AbstractPreAuthenticatedProcessingFilter
- CasAuthenticationFilter
- OAuth2LoginAuthenticationFilter
- Saml2WebSsoAuthenticationFilter
- UsernamePasswordAuthenticationFilter
- OpenIDAuthenticationFilter
- DefaultLoginPageGeneratingFilter
- DefaultLogoutPageGeneratingFilter
- ConcurrentSessionFilter

# Security Filters - II

- DigestAuthenticationFilter
- BearerTokenAuthenticationFilter
- BasicAuthenticationFilter
- RequestCacheAwareFilter
- SecurityContextHolderAwareRequestFilter
- JaasApiIntegrationFilter
- RememberMeAuthenticationFilter
- AnonymousAuthenticationFilter
- OAuth2AuthorizationCodeGrantFilter
- SessionManagementFilter
- ExceptionTranslationFilter
- FilterSecurityInterceptor
- SwitchUserFilter

# Basic Spring Security Configuration with Spring Boot

@Configuration

@EnableWebSecurity

```
public class SecurityConfig {  
    @Bean  
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
        http.csrf().disable()  
            .authorizeRequests().antMatchers("/**").permitAll();  
        return http.build();  
    }  
    @Bean  
    public HttpFirewall getHttpFirewall() {  
        StrictHttpFirewall strictHttpFirewall = new StrictHttpFirewall();  
        strictHttpFirewall.setAllowSemicolon(true);  
        return strictHttpFirewall;  
    }  
    // @Bean  
    // public WebSecurityCustomizer ignoringCustomizer() {  
    //     return (web) -> web.ignoring().antMatchers("/**");  
    // }  
}
```

# More Advanced Spring Security Configuration

@Configuration

@EnableWebSecurity

public class SecurityConfig {

@Autowired

private JwtRequestFilter jwtRequestFilter;

@Autowired

private JwtAuthenticationEntryPoint jwtAuthenticationEntryPoint;

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http.csrf().disable()

.authorizeRequests()

.mvcMatchers(POST, "/api/auth/login", "/api/auth/register").permitAll()

.mvcMatchers(GET, "/api/articles", "/api/articles/\*\*").permitAll()

.mvcMatchers(GET, "/api/users", "/api/users/\*\*").hasRole(ADMIN.name())

.mvcMatchers("/\*\*").hasAnyRole(ADMIN.name(), AUTHOR.name())

.and()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)

.and()

.exceptionHandling().authenticationEntryPoint(jwtAuthenticationEntryPoint);

http.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

return http.build();

}

}

# JwtRequestFilter - I

@Component

@Slf4j

@Order

```
public class JwtRequestFilter extends OncePerRequestFilter {
```

```
    @Autowired
```

```
    private UserService userService;
```

```
    @Autowired
```

```
    private JwtUtils jwtUtils;
```

```
    @Override
```

```
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
```

( - continues on next slide - )

## JwtRequestFilter - II

```
final String authorizationHeader = request.getHeader("Authorization");
String username = null;
String jwtToken = null;
if(authorizationHeader != null) {
    if (authorizationHeader.startsWith("Bearer ")) {
        jwtToken = authorizationHeader.substring(7);
        try {
            username = jwtUtils.getUsernameFromToken(jwtToken);
        } catch (IllegalArgumentException ex) {
            log.error("Unable to get JWT token.");
            throw new BadCredentialsException("Unable to get JWT token.");
        } catch (ExpiredJwtException ex) {
            log.error("JWT token has expired.");
            throw new BadCredentialsException("JWT token has expired.");
        }
    } else {
        log.error("JWT token does not begin with 'Bearer ' prefix.");
        throw new BadCredentialsException("JWT token does not begin with 'Bearer ' prefix.");
    }
}
```

( - continues on next slide - )

# JwtRequestFilter - III

```
if(username != null) {
    User user = userService.findUserByEmail(username);
    if(jwtUtils.validateToken(jwtToken, user)) {
        UsernamePasswordAuthenticationToken authenticationToken =
            new UsernamePasswordAuthenticationToken(user, null, user.getAuthorities());
        authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
        SecurityContextHolder.getContext().setAuthentication(authenticationToken);
    } else {
        throw new BadCredentialsException("JWT token has expired.");
    }
}
filterChain.doFilter(request, response);
}
```



# JwtAuthenticationEntryPoint

@Component

```
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint, Serializable {
```

@Override

```
public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException)
    throws IOException, ServletException {
```

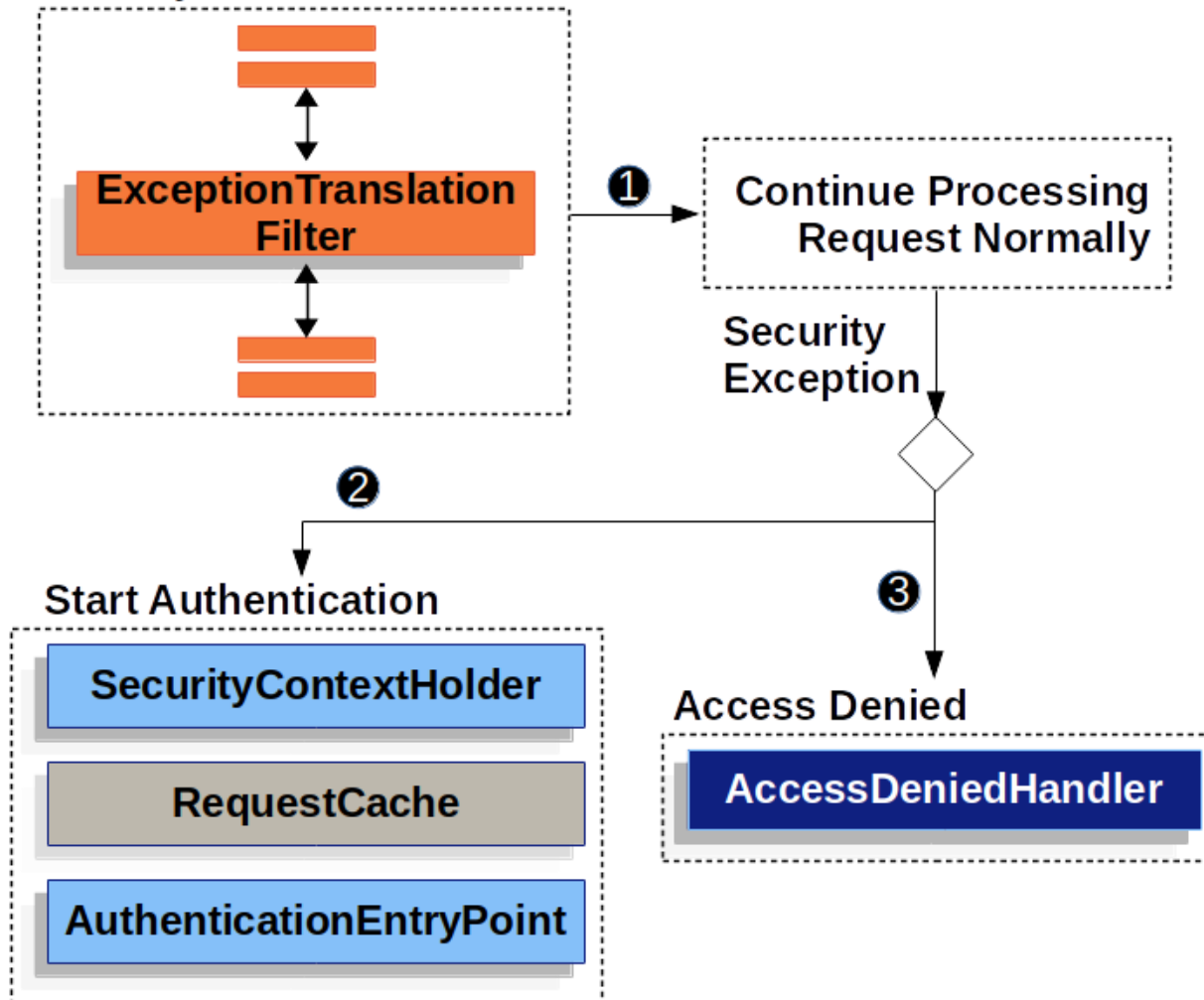
```
    response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized");
```

```
}
```

```
}
```

# DelegatingFilterProxy

## SecurityFilterChain



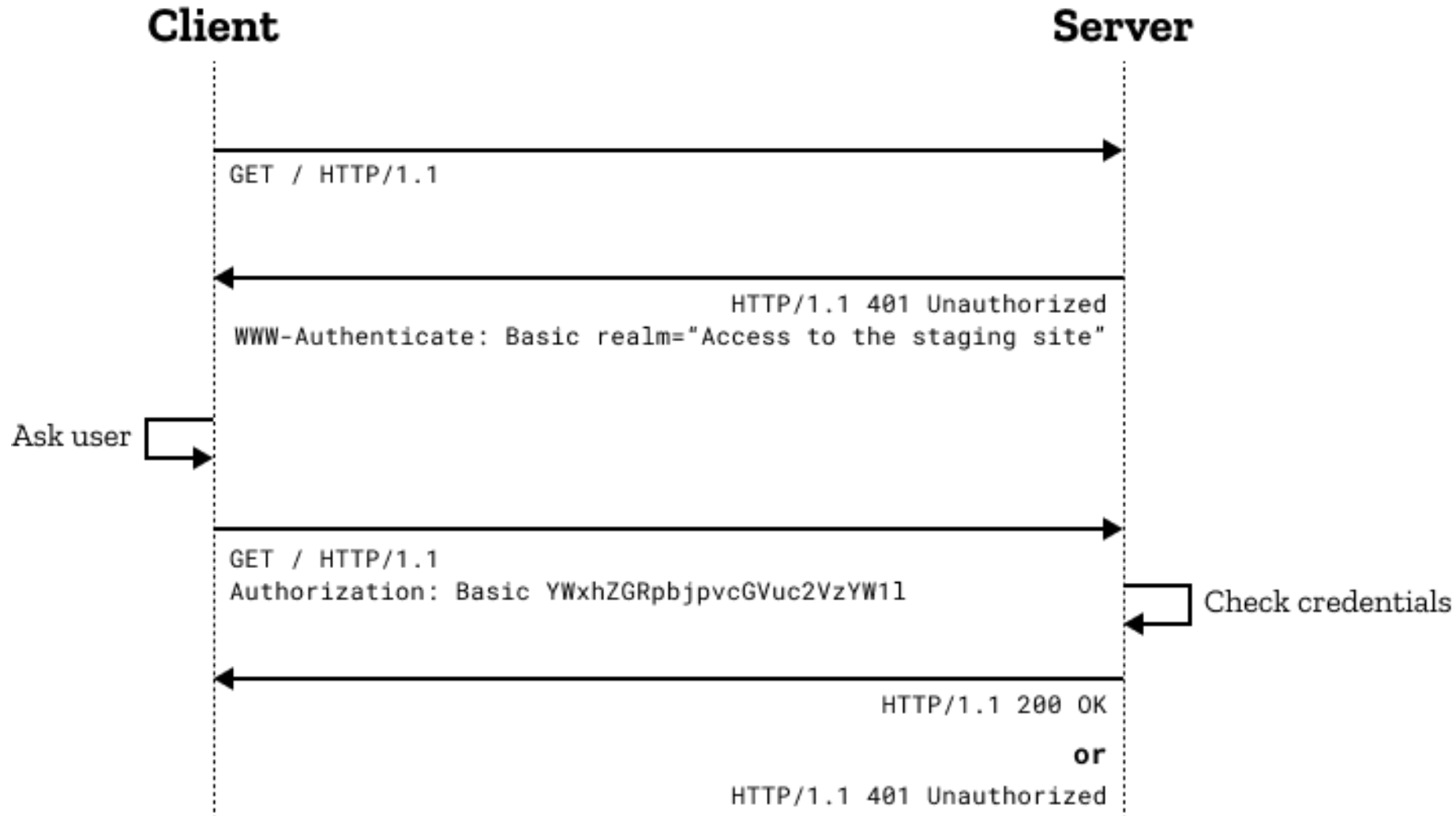
1. First, the [ExceptionTranslationFilter](#) invokes [FilterChain.doFilter\(request, response\)](#) to invoke the rest of the application.
2. If the user is not authenticated or it is an [AuthenticationException](#), then **Start Authentication**.
  - The [SecurityContextHolder](#) is cleared out
  - The [HttpServletRequest](#) is saved in the [RequestCache](#). When the user successfully authenticates, the [RequestCache](#) is used to replay the original request.
  - The [AuthenticationEntryPoint](#) is used to request credentials from the client. For example, it might redirect to a log in page or send a [WWW-Authenticate](#) header.
3. Otherwise if it is an [AccessDeniedException](#), then **Access Denied**. The [AccessDeniedHandler](#) is invoked to handle access denied.

\* If the application does not throw an [AccessDeniedException](#) or an [AuthenticationException](#), then [ExceptionTranslationFilter](#) does not do anything.

# WWW-Authenticate – Authentication Schemes:

- **Basic** - See [RFC 7617](#), base64-encoded credentials. More information below.
- **Bearer** - See [RFC 6750](#), bearer tokens to access OAuth 2.0-protected resources
- **Digest** - See [RFC 7616](#). Firefox 93 and later support the SHA-256 algorithm. Previous versions only support MD5 hashing (not recommended).
- **HOBA** - See [RFC 7486](#), Section 3, **HTTP Origin-Bound Authentication**, digital-signature-based
- **Mutual** - See [RFC 8120](#)
- **Negotiate / NTLM** - See [RFC4599](#)
- **VAPID** - See [RFC 8292](#)
- **SCRAM** - See [RFC 7804](#)
- **AWS4-HMAC-SHA256** - See [AWS docs](#). This scheme is used for AWS3 server authentication.

# WWW-Authenticate – Basic Authentication Scheme



# The OAuth 2.0 Authorization Framework: Bearer Token Usage

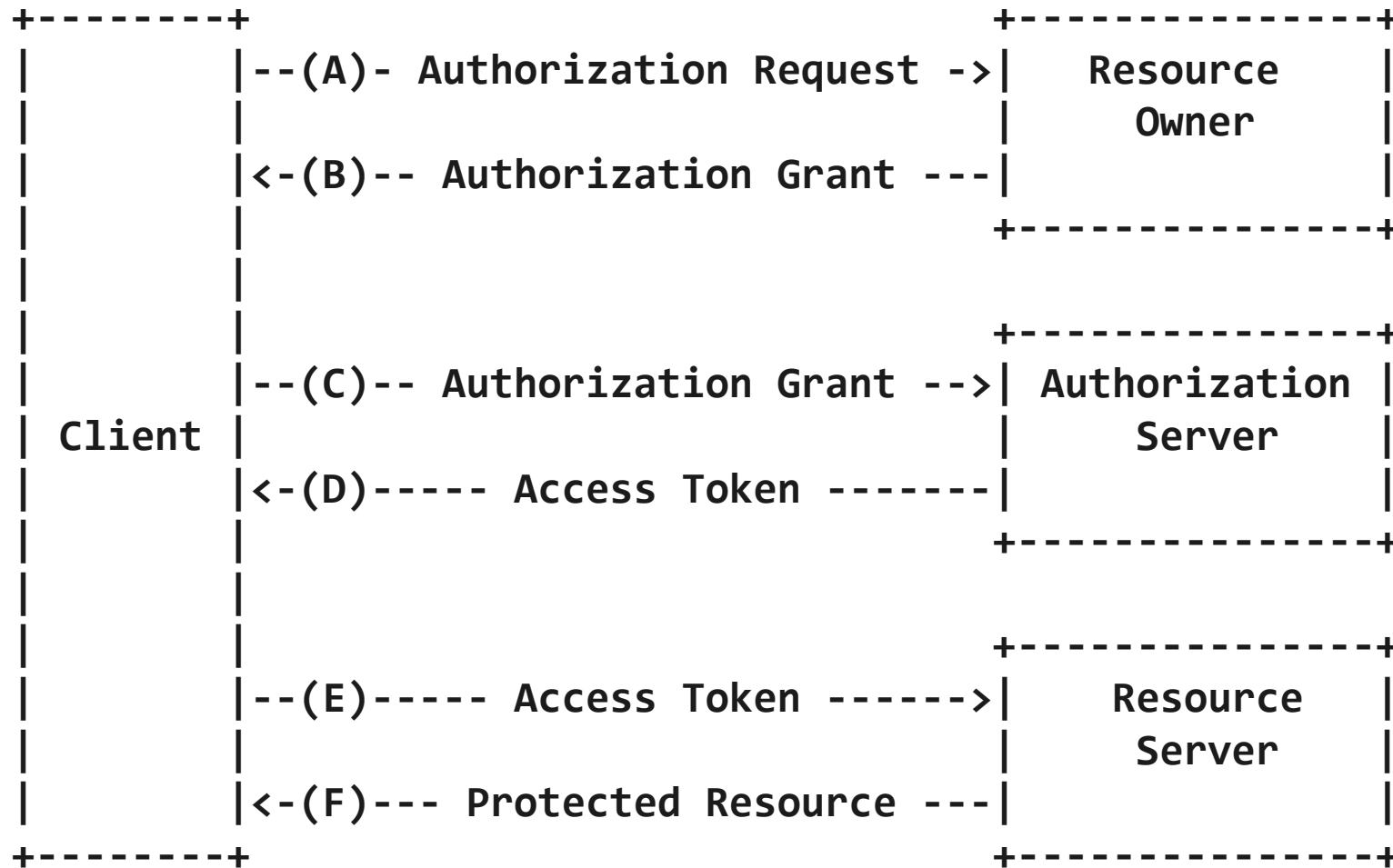


Figure 1: Abstract Protocol Flow

# WWW-Authenticate Response Header Field

- Example - the WWW-Authenticate Response Header Field:

HTTP/1.1 401 Unauthorized

WWW-Authenticate: Bearer realm="example"

- Example - the token expired:

HTTP/1.1 401 Unauthorized

WWW-Authenticate: Bearer realm="example",  
error="invalid\_token",  
error\_description="The access token expired"

# Authorization Request Header Field

- Example Bearer token authentication request:

GET /resource HTTP/1.1

Host: server.example.com

Authorization: Bearer mF\_9.B5f-4.1Jq

# Json Web Token (JWT)

- Specific type of token that can be exchanged using Bearer authentication scheme:

`HMAC_SHA256( secret, base64urlEncoding(header) + '.' + base64urlEncoding(payload) )`

The three parts are encoded separately using [Base64url Encoding RFC 4648](#), and concatenated using periods to produce the JWT:

```
const token = base64urlEncoding(header) + '.' + base64urlEncoding(payload) + '.' +  
base64urlEncoding(signature)
```

- JWT example:

`GET /resource HTTP/1.1`

`Host: server.example.com`

`Authorization: Bearer`

`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJsb2dnZWRRJbkFzIjoieWRtaW4iLCJpYXQiOiE0MjMzNzk2Mzh9.gzS  
raSYS8EXBxLN_oWnFSRgCzcmJmMjLiuyu5CSpyHI`



# Json Web Token (JWT) Standard Claim Fields

Code	Name	Description
<b>Standard claim fields</b>		The internet drafts define the following standard fields ("claims") that can be used inside a JWT claim set.
iss	Issuer	Identifies principal that issued the JWT.
sub	Subject	Identifies the subject of the JWT.
aud	Audience	Identifies the recipients that the JWT is intended for. Each principal intended to process the JWT <b>must</b> identify itself with a value in the audience claim. If the principal processing the claim does not identify itself with a value in the aud claim when this claim is present, then the JWT <b>must</b> be rejected.
exp	Expiration Time	Identifies the expiration time on and after which the JWT <b>must not</b> be accepted for processing. The value must be a NumericDate: <sup>[9]</sup> either an integer or decimal, representing seconds past <a href="https://en.wikipedia.org/wiki/ISO_8601#T_literals">1970-01-01 00:00:00Z</a> .
nbf	Not Before	Identifies the time on which the JWT will start to be accepted for processing. The value must be a NumericDate.
iat	Issued at	Identifies the time at which the JWT was issued. The value must be a NumericDate.
jti	JWT ID	Case-sensitive unique identifier of the token even among different issuers.

# Method Level Security

```
@PreAuthorize("hasRole('USER')")  
public void create(Contact contact);
```

```
@PreAuthorize("hasPermission(#contact, 'admin')")  
public void deletePermission(Contact contact, Sid recipient, Permission permission);
```

```
@PreAuthorize("#c.name == authentication.name")  
public void doSomething(@P("c") Contact contact);
```

```
@PreAuthorize("hasRole('USER')")  
@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject,  
'admin')")  
public List<Contact> getAll();
```

# Thank's for Your Attention!



Trayan Iliev

IPT – Intellectual Products & Technologies

<http://iproduct.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>